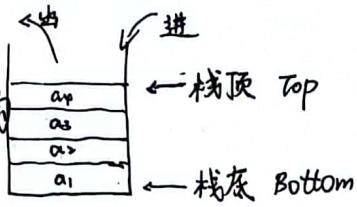
第三章 栈、队列和数组

一. 栈

只允许一端进行插入或删除的线性表.

后进先出 (LIFO)

1. 逻辑结构



不同

n个元素进栈，出栈序列：$\frac{1}{n+1}C_{2n}^{n}$ (卡特兰数)

2. 存储结构

(1) 顺序栈：一组地址连续的存储单元.

```
#define MaxSize 50
typedef struct {
    ElemType data[MaxSize];
    int top;            //栈底指针
} SqStack;
```

① 初始化：

```
void InitStack(SqStack &S){
    S.top = -1;
}
```

② 判空：

```
bool StackEmpty(SqStack S){
    if(S.top == -1)  return true;
    else             return false;
}
```

可能发生栈上溢

③ 进栈：

```
bool Push(SqStack &S, ElemType x){
    if(S.top == MaxSize-1)  return false;
    S.data[++S.top] = x;
    return true;
}
```

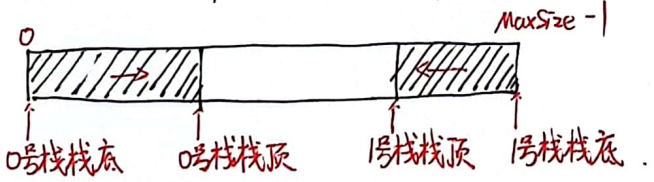
④ 出栈：

```
bool Pop(SqStack &S, ElemType &x){
    if(S.top == -1)  return false;
    x = S.data[S.top--];
    return true;
}
```

⑤ 读取栈顶元素：

```
bool GetTop(SqStack &S, ElemType &x){
    if(S.top == -1)  return false;
    x = S.data[S.top];
    return true;
}
```

(2) 共享栈：两个顺序栈共享一个一维数组空间. 栈底位置相对不变的特性



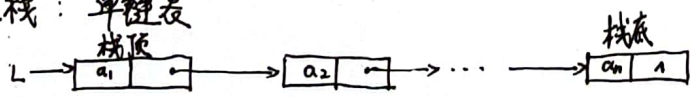
① top0 = -1，0号空
top1 = MaxSize，1号空

② top1 - top0 = 1，栈满

③ 0号进栈，先加1，后赋值
1号进栈，先减1，后赋值

更有效利用存储空间.

(3) 链栈：单链表



① 规定操作在表头进行.

② 没有头结点.

③ 便于多个栈共享存储空间和提高效率，不存在栈满上溢

3. 栈的应用

(1) 括号匹配：

① 初始设置一个空栈，顺序读入括号；

② 若是右括号，置于栈顶；

③ 若是左括号，压入栈中；

④ 算法结束，栈为空，否则括号序列不匹配。

(2) 算术表达式

(1) 中缀表达式转后缀表达式：借助栈保存暂时不能确定运算顺序的运算符。

① 遇操作数，加入后缀表达式

② 遇界限符："(" 入栈；")" 依次弹出运算符直到 "("。括号不加入后缀表达式。

③ 遇运算符：优先级高于除 "(" 外的栈顶运算符，入栈；
否则，弹≥当前运算符的运算符，直到＜当前运算符或 "("。

手算方法：

① 按照运算符运算顺序对所有运算单位加括号；

② 运算符移至对应括号后面，"左操作数 右操作数 运算符" 重新组合。

③ 去括号。

$A+B*(C-D)-E/F$

① $((A+_{②}(B*_{③}(C-_{④}D)))-_{⑤}(E/_{⑥}F))$

② $((A(B(CD)-_{①})*_{②})+_{④}(EF)/_{⑥})-_{⑤}$

③ $ABCD-*+EF/-$

(2) 后缀表达式求值

从左往右扫描，若操作数，压入栈中；若操作符，栈中退 x 和 y，形成 y<op>x，将结果压栈。

第二个 第一个

(3) 递归

① 在递归调用的过程中，系统为每一层的返回点、局部变量、传入实参等开辟了递归工作栈来进行数据存储。

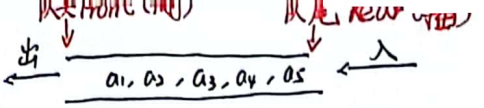
② 次数过多 → 栈溢出

③ 效率过高：包含很多重复计算。

④ 递归算法 → 非递归，借助栈。

一. 队列

1. 逻辑结构



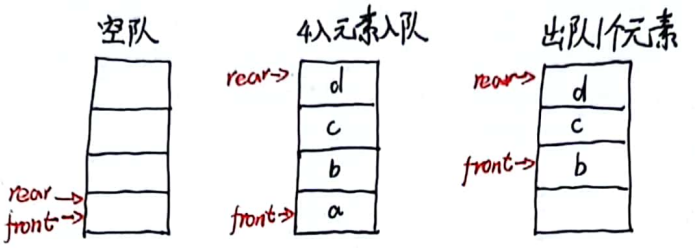
① 只允许在表的一端插入，另一端删除。

② 先进先出。

2. 存储结构

(1) 顺序队列

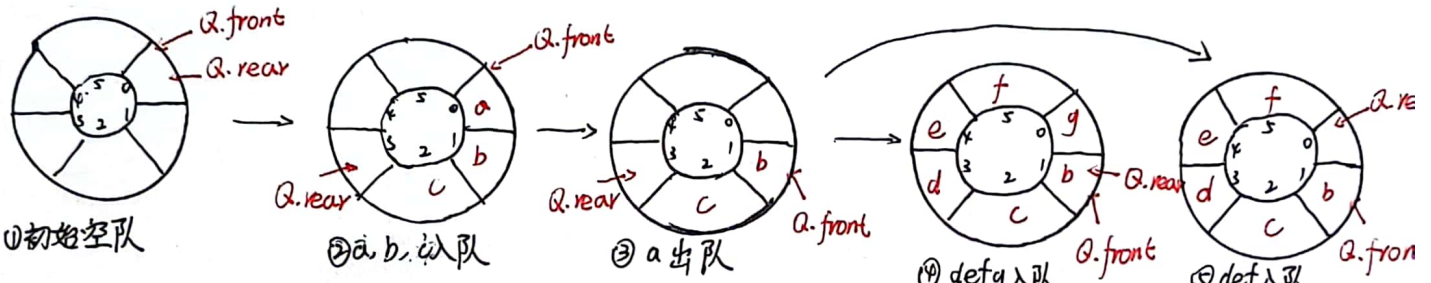
```
#define  MaxSize  50
  typedef  struct {
     ElemType  data[MaxSize];
     int  front, rear;
  } SqQuene
```



队满：Q.rear = MaxSize (不能作为队满条件，假溢出)

(只有一个元素时仍满)

(2) 循环队列：将存储队列元素的表从逻辑上视为一个环。



① 初始：Q.front = Q.rear = 0

② 入队，队尾+1：Q.rear = (Q.rear + 1) % MaxSize

③ 出队，队首+1：Q.front = (Q.front + 1) % MaxSize

④ 队列长：(Q.rear + MaxSize - Q.front) % MaxSize ☆

⑤ 队空：Q.front == Q.rear

⑥ 队满：(Q.rear + 1) % MaxSize == Q.front.

△ 如何区分循环队列空/满的判断条件？

方法一：牺牲一个单元，约定以"队头指针在队尾指针的下一位置作为队满标志"，如上图。

方法二：类型中增设 size 表示元素个数 { 队空：Q.size == 0；队满：Q.size == MaxSize }

方法三：类型中增设 tag { 队空：tag = 0 且因删除导致 Q.front == Q.rear；队满：tag = 1 且因插入导致 Q.front == Q.rear }

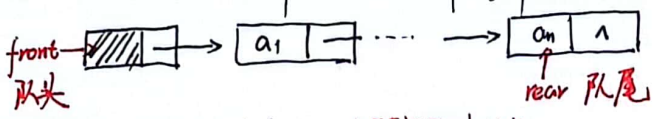
入队：

```
bool EnQueue(SqQueue &Q, ElemType x){
    if ((Q.rear+1) % MaxSize == Q.front)
        return false;
    Q.data[Q.rear] = x;
    Q.rear = (Q.rear+1) % MaxSize;
    return true;
}
```

出队：

```
bool DeQueue(SqQueue &Q, ElemType &x){
    if (Q.rear == Q.front)  return false;
    x = Q.data[Q.front];
    Q.front = (Q.front+1) % MaxSize;
    return true;
}
```

(3) 链式队列：同时有队头指针和队尾指针的单链表。



①头指针指向队头结点
②尾指针指向队尾结点
③删除通常只修改头指针
④尾插
⑤适合数据元素变动比较大的情形，而且不存在队列满产生溢出的问题。

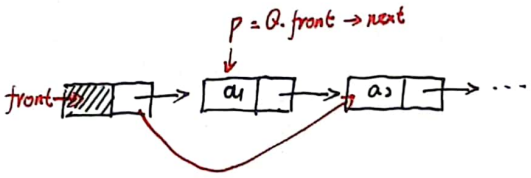
```
typedef struct LinkNode{        //链式队列结点
    ElemType data;
    struct LinkNode *next;
}LinkNode;
typedef struct{                 //链式队列
    LinkNode *front, *rear;
}LinkQueue;
```

① 初始化：
```
void InitQueue(LinkQueue &Q){
    Q.front = Q.rear = (LinkNode*)malloc(sizeof(LinkNode));
    Q.front->next = NULL;
}
```

② 判空：
```
bool IsEmpty(LinkQueue Q){
    if(Q.front == Q.rear)   return true;
    else return false;
}
```

③ 入队：
```
void EnQueue(LinkQueue &Q, ElemType x){
    LinkNode *s = (LinkNode*)malloc(sizeof(LinkNode));
    s->data = x;
    s->next = NULL;
    Q.rear->next = s;           //插入链尾
    Q.rear = s;                 //修改尾指针
}
```

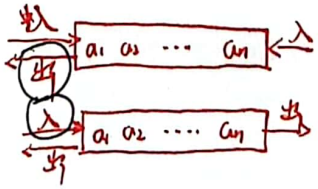
④ 出队：
```
bool DeQueue(LinkQueue &Q, ElemType &x){
    if(Q.front == Q.rear)   return false;
    LinkNode *p = Q.front->next;
    x = p->data;
    Q.front->next = p->next;
    if(Q.rear == p)   Q.rear = Q.front;   //原队列中只有一个结点，删除后变空
    free(p);
    return true;
}
```



(4) 双端队列：逻辑结构仍是线性结构

①输出受限：一端插、删，另一端只插

②输入受限：一端插、删，另一端只删。

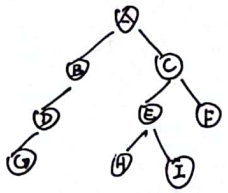


输出固定，看输入

输入固定，看输出

队列应用

(1) 层次遍历



使用队列是为了保存下一步的处理顺序.

		队内	队外
1	A入	A	
2	A出, BC入	BC	A
3	B出, D入	CD	AB
4	C出, EF入	DEF	ABC
5	D出, G入	EFG	ABCD
6	E出, HI入	FGHI	ABCDE
7	F出	GHI	ABCDEF
8	GHI出		ABCDEFGHI

①根结点入队

②队空, 结束; 否则重复③

③队列中第一个结点出队, 并访问。有左, 将左入队; 有右, 将右入队, 返回②

(2) 在计算机系统中

①解决主机与外部设备之间速度不匹配的问题 (如打印机与主机, 设置一个打印数据缓冲区).

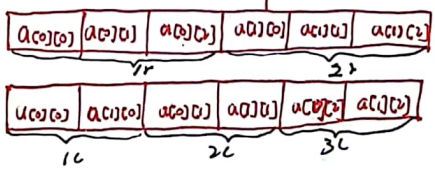
②解决由多用户引起的资源竞争问题 (如CPU资源的竞争).

③页面替换 FIFO 算法.

三. 数组和特殊矩阵

1. 数组: 由n个相同类型的数据元素构成的有限序列. { 一旦被定义, 其维数和维界不再改变; 只有存取和修改元素操作.

2. 数组的存储结构: 多维映射一维 { 行优先: 先行后列; 列优先: 先列后行



3. 特殊矩阵相关概念

(1) 压缩存储: 为多个值相同的元素只分配一个存储空间, 对零元素不分配空间.

(2) 特殊矩阵: ① 对称矩阵: $a_{ij}=a_{ji}$ $(1\le i,j\le n)$

n阶对称矩阵A (i,j从1开始) → 一维B (下标从0开始) $B[\frac{n(n+1)}{2}]$

$$k=\begin{cases}\frac{i(i-1)}{2}+j-1, & i\ge j\\ \frac{j(j-1)}{2}+i-1, & i<j\end{cases}$$

② 三角矩阵: 上三角区元素为同一常量为下三角矩阵

$$k=\begin{cases}\frac{i(i-1)}{2}+j-1, & i\ge j\\ \frac{n(n+1)}{2}, & i<j\end{cases}$$

按行优先

$B[\frac{n(n+1)}{2}+1]$ 多一个常量.

下三角区元素为同一常量为上三角矩阵

$$\begin{bmatrix} a_{1,1} & a_{1,2} & \cdots & a_{1,n}\\ & a_{2,2} & \cdots & a_{2,n}\\ & & \ddots & \vdots\\ & & & a_{n,n}\end{bmatrix}$$

$$k=\begin{cases}\frac{(n+n-i+1)\times i}{2}+j-i, & i\le j\\ \frac{n(n+1)}{2}, & i>j\end{cases}$$

前i-1行 $\frac{(i-1)(2n-i+2)}{2}$ 第i行所在位置 $+j-i$

③ 三对角矩阵: 当$|i-j|>1$时, $a_{ij}=0$ $(1\le i,j\le n)$

已知A → 一维B (从0开始): $k=2i+j-3$

前i-1行 $3(i-1)-1$ + $j-i+2$ − 前i-1行 $3(i-1)+j-i+2$

已知B → 三对角A: $i=\lfloor(k+1)/3+1\rfloor$, $j=k-2i+3$

前i-1行共$3(i-1)$个

前i行共$3i-1$个

$3(i-1)-1\le k<3i-1$

$\Rightarrow i\le\lfloor(k+1)/3+1\rfloor$

④稀疏矩阵:

非0元素非常少的矩阵.

~~压缩存储~~

将非零元素及其行、列构成一个三元组并保存稀疏矩阵的行数、列数和非零元素的个数.

失去了随机存取的特性

三元组或十字链表

1. 栈和队列逻辑结构相同，都属于线性结构，只是对数据的运算不同。
2. 上溢是指存储器满，还往里写；下溢是指存储器空，还往外读。
3. 链式队列长度受内存空间限制，不能根据头指针和尾指针计算队列元素个数。
4. 顺序、链式队列进队、出队时间均为O(1)。
5. 循环队列是指顺序存储的队列，而不是指逻辑上的循环，如循环单链表表示的队列不能称为循环队列。
6. 消除递归不一定使用栈，如单向递归和尾递归可用迭代的方式消除。
7. 通常使用栈来处理函数或过程调用。